MTI #1666
Docket No. 18153.0035

LOW OVERHEAD INTERRUPT

FIELD OF THE INVENTION:

The present invention relates to systems and methods for instruction processing and, more particularly, to systems and methods for providing efficient interrupt processing when exceptions to normal instruction processing occur during program execution.

BACKGROUND OF THE INVENTION:

Processors, including microprocessors, digital signal processors and microcontrollers, operate by running software programs that are embodied in one or more series of instructions stored in a memory. The processors run the software by fetching the instructions from the series of instructions, decoding the instructions and executing them. The instructions themselves control the order in which the processor fetches and executes the instructions. For example, the order for fetching and executing each instruction may be inherent in the order of the instructions within the series. Alternatively, instructions such as branch instructions, conditional branch instructions, subroutine calls and other flow control instructions may cause instructions to be fetched and executed out of the inherent order of the instruction series.

When a processor fetches and executes instructions in the inherent order of the instruction series, the processor may execute the instructions very efficiently without wasting processor cycles to determine, for example, where the next instruction is. When flow control instructions are processed, one or more processor cycles may be wasted while the processor locates and fetches the next instruction required for execution.

Another inefficient process conventionally is the process of handling an interrupt when exceptions to normal instruction processing occur during program execution. There are many

conditions that may trigger an interrupt including, for example, a processor reset, an oscillator fail condition, a stack overflow condition, an address error condition, an illegal instruction condition, an arithmetic error condition and various priority interrupts for giving effect to various input/output, program or other processes. When any or all of these conditions occur within a processor, an interrupt flag is set. The processor then determines the priority level of the interrupt and the priority level of the processor. When the priority level of one or more of the interrupts is greater than the priority level of the processor, then the interrupt is serviced.

In servicing an interrupt, the processor fetches an interrupt service routine (ISR) and executes it. The ISR is a series of instructions stored somewhere in memory. Accordingly, fetching the ISR is equivalent to the processor executing a flow control instruction in that it is conventionally equally wasteful of processor cycles. Conventionally, for example, servicing an interrupt flushes instructions being processed from the instruction register and requires one, two or more cycles for the first instruction of the interrupt service routine to be loaded into the instruction register depending on where the ISR is stored in memory. Similarly, after servicing an interrupt, one or two cycles are required to reset the program counter to its original state prior to the ISR, fetch and begin executing instructions again.

Accordingly, there is a need for a processor that reduces or eliminates processor cycle losses during interrupt handling due to delays associated with a change in instruction flow. There is still a further need to provide an interrupt capability to handle exceptions that arise during the processing of a repeat instruction loop. There is a further need for nested interrupts to be processed in an efficient manner. There is still a further need for instruction processing to resume with a minimum loss in processor cycles after an interrupt is processed. There is still a further need to provide a solution for handling interrupts that minimizes the amount of logic

required to implement the solution and thus the amount of space required on the processor to support the solution.

SUMMARY OF THE INVENTION:

According to the present invention, a method and processor for interrupt processing to save processor cycles during the handling of interrupts. More particularly, upon an interrupt, the first instruction from an interrupt service routine (ISR) is loaded into an instruction register for immediate execution to save at least one cycle of interrupt instruction fetching. Simultaneously, the address of the second instruction from the ISR is stored into a program counter. Also, the next instruction in the regular program cycle for execution is taken from a prefetch register and stored in a holding register (or to the stack) and the rest of the machine state may be shadowed. Subsequently, the second instruction from the ISR is fetched and executed and the interrupt is serviced. When finished, the next regular instruction for execution is loaded into the prefetch register from the holding register (or stack) for subsequent execution and the rest of the machine state is restored from the stack or shadow registers. In the next cycle, the instruction from the holding register is executed and the following instruction is fetched into the prefetch. These steps save at least one cycle in processing interrupts.

According to an embodiment of the invention, a method for processing a low-overhead interrupt includes detecting the occurrence of an interrupt condition that requires servicing and determining that the interrupt condition corresponds to a fast interrupt. When the interrupt condition corresponds to a fast interrupt, a first instruction of an interrupt service routine (ISR) is loaded into an instruction register for immediate execution. In addition, an address of a second instruction within the ISR is loaded into a program counter. Subsequently, the second ISR

instruction is fetched while executing the first ISR instruction. The method may also include storing contents of a prefetch register into a holding register based on determining that the interrupt is a fast interrupt. The method may also include storing a program counter value and a status register value onto a stack or shadow registers along with the rest of the machine state.

According to another embodiment of the invention, the remaining instructions in the ISR are executed and control returns from the ISR. The return of control may include restoring an instruction from the holding register into the prefetch register. The return of control may further include popping a program counter value and a status register value from a stack or shadow registers and storing them into respective program counter and status registers. The method may further include executing the instruction restored to the prefetch register and fetching the next instruction for execution based on the status register and the program counter register. When the instruction returned to from the interrupt is an instruction within a repeat loop, the status register indicates that a repeat instruction is being processed. Accordingly a loop counter may be decremented after executing the instruction restored to the prefetch register until the repeat loop finishes.

According to another embodiment of the present invention, a processor for handling low-overhead interrupts includes a first interrupt instruction register, a holding register, an interrupt vector register, a program counter and interrupt logic. The first interrupt instruction register stores the first interrupt instruction of each a plurality of interrupt service routines. The interrupt logic is coupled to the registers. Upon an interrupt, the interrupt logic a) loads a first instruction of an interrupt service routine (ISR) from the first interrupt instruction register into an instruction register for immediate execution and b) loads an address of a second instruction within the ISR into the program counter based on the interrupt vector register and executes the ISR.

BRIEF DESCRIPTION OF THE FIGURES:

The above described features and advantages of the present invention will be more fully appreciated with reference to the detailed description and appended figures in which:

Fig. 1 depicts a functional block diagram of an embodiment of a processor chip within which embodiments of the present invention may find application.

Fig. 2 depicts a functional block diagram of a data busing scheme for use in a processor, which has a microcontroller and a digital signal processing engine, within which embodiments of the present invention may find application.

Fig. 3 depicts a functional block diagram of a processor configuration for processing an interruptible repeat according to an embodiment of the present invention.

Fig. 4 depicts a method of processing an interruptible repeat instruction according to an embodiment of the present invention.

Fig. 5 depicts a functional block diagram of a processor configuration for implementing a low over-head interrupt according to an embodiment of the present invention.

Figs. 6A and 6B depict a method of processing an interrupt in an efficient manner according to an embodiment of the present invention.

DETAILED DESCRIPTION:

According to the present invention, a method and processor for interrupt processing to save processor cycles during the handling of interrupts. More particularly, upon an interrupt, the first instruction from an interrupt service routine (ISR) is loaded into an instruction register for immediate execution to save at least one cycle of interrupt instruction fetching. Simultaneously,

the address of the second instruction from the ISR is stored into a program counter. Also, the next instruction in the regular program cycle for execution is taken from a prefetch register and stored in a holding register (or to the stack). Subsequently, the second instruction from the ISR is fetched and executed and the interrupt is serviced. When finished, the next regular instruction for execution is loaded into the prefetch register from the holding register (or stack) for subsequent execution. The program counter and a status register are restored from the stack or a shadow register. In the next cycle, the instruction from the holding register is executed and the following instruction is fetched into the prefetch. These steps save at least one cycle in processing interrupts.

In order to describe embodiments of interrupt processing, an overview of pertinent processor elements is first presented with reference to Figs. 1 and 2. The interrupt processing is then described, along with details of servicing an interrupt during a repeat instruction loop, more particularly with reference to Figs. 3-6B.

Overview of Processor Elements

Fig. 1 depicts a functional block diagram of an embodiment of a processor chip within which the present invention may find application. Referring to Fig. 1, a processor 100 is coupled to external devices/systems 140. The processor 100 may be any type of processor including, for example, a digital signal processor (DSP), a microprocessor, a microcontroller or combinations thereof. The external devices 140 may be any type of systems or devices including input/output devices such as keyboards, displays, speakers, microphones, memory, or other systems which may or may not include processors. Moreover, the processor 100 and the external devices 140 may together comprise a stand alone system.

The processor 100 includes a program memory 105, an instruction fetch/decode unit 110, instruction execution units 115, data memory and registers 120, peripherals 125, data I/O 130, and a program counter and loop control unit 135. The bus 150, which may include one or more common buses, communicates data between the units as shown.

The program memory 105 stores software embodied in program instructions for execution by the processor 100. The program memory 105 may comprise any type of nonvolatile memory such as a read only memory (ROM), a programmable read only memory (PROM), an electrically programmable or an electrically programmable and erasable read only memory (EPROM or EEPROM) or flash memory. In addition, the program memory 105 may be supplemented with external nonvolatile memory 145 as shown to increase the complexity of software available to the processor 100. Alternatively, the program memory may be volatile memory which receives program instructions from, for example, an external non-volatile memory 145. When the program memory 105 is nonvolatile memory, the program memory may be programmed at the time of manufacturing the processor 100 or prior to or during implementation of the processor 100 within a system. In the latter scenario, the processor 100 may be programmed through a process called in-line serial programming.

The instruction fetch/decode unit 110 is coupled to the program memory 105, the instruction execution units 115 and the data memory 120. Coupled to the program memory 105 and the bus 150 is the program counter and loop control unit 135. The instruction fetch/decode unit 110 fetches the instructions from the program memory 105 specified by the address value contained in the program counter 135. The instruction fetch/decode unit 110 then decodes the fetched instructions and sends the decoded instructions to the appropriate execution unit 115.

The instruction fetch/decode unit 110 may also send operand information including addresses of data to the data memory 120 and to functional elements that access the registers.

The program counter and loop control unit 135 includes a program counter register (not shown) which stores an address of the next instruction to be fetched. During normal instruction processing, the program counter register may be incremented to cause sequential instructions to be fetched. Alternatively, the program counter value may be altered by loading a new value into it via the bus 150. The new value may be derived based on decoding and executing a flow control instruction such as, for example, a branch instruction. In addition, the loop control portion of the program counter and loop control unit 135 may be used to provide repeat instruction processing and repeat loop control as further described below.

The instruction execution units 115 receive the decoded instructions from the instruction fetch/decode unit 110 and thereafter execute the decoded instructions. As part of this process, the execution units may retrieve one or two operands via the bus 150 and store the result into a register or memory location within the data memory 120. The execution units may include an arithmetic logic unit (ALU) such as those typically found in a microcontroller. The execution units may also include a digital signal processing engine, a floating point processor, an integer processor or any other convenient execution unit. A preferred embodiment of the execution units and their interaction with the bus 150, which may include one or more buses, is presented in more detail below with reference to Fig. 2.

The data memory and registers 120 are volatile memory and are used to store data used and generated by the execution units. The data memory 120 and program memory 105 are preferably separate memories for storing data and program instructions respectively. This format is a known generally as a Harvard architecture. It is noted, however, that according to the

present invention, the architecture may be a Von-Neuman architecture or a modified Harvard architecture which permits the use of some program space for data space. A dotted line is shown, for example, connecting the program memory 105 to the bus 150. This path may include logic for aligning data reads from program space such as, for example, during table reads from program space to data memory 120.

Referring again to Fig. 1, a plurality of peripherals 125 on the processor may be coupled to the bus 125. The peripherals may include, for example, analog to digital converters, timers, bus interfaces and protocols such as, for example, the controller area network (CAN) protocol or the Universal Serial Bus (USB) protocol and other peripherals. The peripherals exchange data over the bus 150 with the other units.

The data I/O unit 130 may include transceivers and other logic for interfacing with the external devices/systems 140. The data I/O unit 130 may further include functionality to permit in circuit serial programming of the Program memory through the data I/O unit 130.

Fig. 2 depicts a functional block diagram of a data busing scheme for use in a processor 100, such as that shown in Fig. 1, which has an integrated microcontroller arithmetic logic unit (ALU) 270 and a digital signal processing (DSP) engine 230. This configuration may be used to integrate DSP functionality to an existing microcontroller core. Referring to Fig. 2, the data memory 120 of Fig. 1 is implemented as two separate memories: an X-memory 210 and a Y-memory 220, each being respectively addressable by an X-address generator 250 and a Y-address generator 260. The X-address generator may also permit addressing the Y-memory space thus making the data space appear like a single contiguous memory space when addressed from the X address generator. The bus 150 may be implemented as two buses, one for each of the X and Y memory, to permit simultaneous fetching of data from the X and Y memories.

The W registers 240 are general purpose address and/or data registers. The DSP engine 230 is coupled to both the X and Y memory buses and to the W registers 240. The DSP engine 230 may simultaneously fetch data from each the X and Y memory, execute instructions which operate on the simultaneously fetched data and write the result to an accumulator (not shown) and write a prior result to X or Y memory or to the W registers 240 within a single processor cycle.

In one embodiment, the ALU 270 may be coupled only to the X memory bus and may only fetch data from the X bus. However, the X and Y memories 210 and 220 may be addressed as a single memory space by the X address generator in order to make the data memory segregation transparent to the ALU 270. The memory locations within the X and Y memories may be addressed by values stored in the W registers 240.

Any processor clocking scheme may be implemented for fetching and executing instructions. A specific example follows, however, to illustrate an embodiment of the present invention. Each instruction cycle is comprised of four Q clock cycles Q1 – Q4. The four phase Q cycles provide timing signals to coordinate the decode, read, process data and write data portions of each instruction cycle.

According to one embodiment of the processor 100, the processor 100 concurrently performs two operations – it fetches the next instruction and executes the present instruction. Accordingly, the two processes occur simultaneously. The following sequence of events may comprise, for example, the fetch instruction cycle:

Q1: Fetch Instruction
Q2: Fetch Instruction
Q3: Fetch Instruction
Q4: Latch Instruction into prefetch register, Increment PC

The following sequence of events may comprise, for example, the execute instruction cycle for a single operand instruction:

Q1: latch instruction into IR, decode and determine addresses of operand data
Q2: fetch operand
Q3: execute function specified by instruction and calculate destination address for data
Q4: write result to destination

The following sequence of events may comprise, for example, the execute instruction cycle for a dual operand instruction using a data pre-fetch mechanism. These instructions pre-fetch the dual operands simultaneously from the X and Y data memories and store them into registers specified in the instruction. They simultaneously allow instruction execution on the operands fetched during the previous cycle.

Q1: latch instruction into IR, decode and determine addresses of operand data
Q2: pre-fetch operands into specified registers, execute operation in instruction
Q3: execute operation in instruction, calculate destination address for data
Q4: complete execution, write result to destination

Repeat Instruction Processing

Fig. 3 depicts a functional block diagram of a configuration for processing an interruptible repeat instruction according to an embodiment of the present invention.

Referring to Fig. 3, the program memory 300 stores program instructions for execution. The program memory 300 includes a repeat instruction, 301, followed by a target instruction 302 for repetition. The repeat instruction may follow the target instruction if a register is implemented to hold instructions after their execution. A prefetch register 305 fetches the next instruction for execution based on the instruction within the program memory pointed to by the program counter (PC) 350. The instruction register 315 stores the current instruction that the processor is executing. The instruction register may be loaded during Q1 with the instruction

from the prefetch register 305 fetched in the previous cycle. The instruction decoder 325 decodes the instruction and provides the decoded instruction and operand addresses to the appropriate execution unit 360. The appropriate execution unit may be, for example, the ALU 270 or the DSP engine 230.

In the case of a repeat instruction, the decoder may provide the decoded instruction and any operand addresses to the loop control unit 330. There are two main types of REPEAT instruction that may be implemented. The first is a REPEAT instruction where the number of iterations for the loop is specified as a loop count value in an immediate operand within the REPEAT instruction itself. The number of bits allocated for the loop count value determines how large the loop count value may be. The second type of REPEAT instruction is a REPEAT W instruction where W stands for W register. According to the REPEAT W instruction, the instruction specifies a W register by including its address within the instruction. The specified W register stores the loop count value for the repeat loop.

The loop control unit 330 executes the REPEAT and REPEAT W instructions when decoded by the decode 325. Specifically, the loop control unit 330 recognizes the REPEAT instruction and sends a control signal to the program counter 350 causing the PC not to increment for all subsequent iterations of the instruction to be repeated until the last one is encountered. In addition, the program fetch control is also inhibited during this time, freeing up the program memory for data access during these cycles. Accordingly, the PC points to the target instruction 302 following the REPEAT instruction during this time.

The loop control unit 330 further sets a repeat status bit 345 which may be one of many bits within a larger status register for the processor. The repeat status bit indicates that the processor is processing a repeat instruction. The repeat status bit is an input to the program

counter and inhibits the incrementing of the program counter and the fetching of instructions when set. The status bit may also be read automatically or otherwise during an interrupt service routine so that program flow control, upon returning from an interrupt service routine, is determined by the loop control unit 330 to resume execution of the repeat loop.

The loop control unit 330 stores the loop control value into the RCOUNT register 340. When the REPEAT instruction includes the loop control value as an immediate operand, the loop control value is written directly into the RCOUNT register 340. When the REPEAT W instruction is used, the loop control value is written from the repeat value register 335, specified by the address in the REPEAT W instruction into the RCOUNT register 340.

When the processor executes the REPEAT instruction for the first time, the instruction following the REPEAT instruction is stored into the prefetch register 305. This instruction is the target instruction 302 which will be repeated. Upon the next processor cycle, the REPEAT status bit (now set) prevents the program counter 350 from incrementing and prevents an instruction fetch from program memory from occurring. Consequently, the pre-fetch register 305 is not overwritten. The instruction register 315 is therefore repeatedly loaded with the contents of the pre-fetch register 305 during each subsequent iteration of the repeat loop.

To test whether the appropriate number of loops have been made, the loop control unit 330 decrements the RCOUNT register and compares its value with a predetermined value which may be, for ex., 0. When the RCOUNT reaches this value, then the loop control unit clears the repeat status bit which will allow the program counter 350 to increment during the last repeat loop iteration. Normal instruction processing resumes for all subsequently fetched instructions. If RCOUNT $\neq$ the predetermined value, then the target instruction is repeated again and the repeat loop remains active until RCOUNT = 0. It will be understood that any logical operation

may be used to test whether the loop is finished. For example, RCOUNT may be incremented instead of decremented and in either case the incrementing/decrementing may be made prior to or after the comparison with a predetermined value. In addition, the predetermined value may be a value other than 0.

At any time during the processing of the repeat instruction, an interrupt may occur and be taken. In order to preserve the target instruction 302 while the interrupt is serviced, the target instruction from the prefetch register 305 may be stored in a holding register 310. The holding register may hold a single value only. Alternatively, the holding register may hold a stack of values thus permitting nested interrupts to be serviced. This hardware last in first out (LIFO) stack would permit nested interrupts to execute REPEAT instructions within their respective ISR, without the need to save and retrieve the contents of the instruction prefetch register 305.

To speed processing of an interrupt service routine, the first interrupt instruction 303 from an interrupt service routine (ISR) may be stored in register 320 and loaded into the instruction register 315 automatically upon detection of the interrupt. This allows the processor 100 to immediately decode and execute the first instruction of the ISR. Simultaneously, PC is saved in the PC shadow register 351, status register (SR) is saved in the SR shadow register, and RCOUNT is saved in the RCOUNT shadow register, and the ISR Vector Register may update the program counter with the address of the second instruction of the ISR. This forces the prefetch register to fetch the second instruction of the ISR while the first instruction is executing. The PC shadow register may hold a single value only. Alternatively, the PC shadow register may hold a stack of values for nested interrupts. The same is true for the RCOUNT shadow register. These measures allow interrupts to be serviced during repeat instruction processing and reduce or eliminate dormant instruction cycles during servicing interrupts.

After an interrupt is serviced, the target instruction 302 from the holding register 310 may be restored to the prefetch register 305 thus permitting repetition of the target instruction to resume. This also reduces or eliminates dormant instruction cycles because the target instruction 302 is restored without the processor having to first reset the PC to read the target instruction from the program memory.

Fig. 4 depicts a method of processing an interruptible repeat instruction according to an embodiment of the present invention. Referring to Fig. 4, in step 400, the processor fetches a repeat instruction. Then in step 405, the processor 100 fetches the target instruction for repetition. In step 410, the processor sets the status register to reflect on-going repeat instruction processes. In step 415, the processor stores a loop count value into a repeat count register. The processor may obtain the loop count value directly from immediate data within the repeat instruction or from a register or other memory location. In the latter scenario, the repeat instruction itself may specify the address of the register or memory location that includes the loop count value.

In step 420, the processor sends a control signal to the program counter 350 to prevent the PC from incrementing. Then in step 425, the processor decrements the repeat count register 340. In step 425, the processor determines whether or not the repeat count register 340 is less than zero. If so, then the repeat loop has finished. Accordingly, in step 430, the processor increments the program counter, resets the repeat status register 345 and fetches the next instruction. If not, then step 440 begins. In step 440, if there is an interrupt, the processor processes the interrupt in step 445. If there is not an interrupt, then the processor executes the instruction to be repeated in step 450. Subsequently, step 425 begins again. In this manner, the

procedure provides and efficiently low overhead interruptible repeat instruction processing capability.

Fig. 5 depicts a functional block diagram of logic and registers for implementing a low overhead interrupt according to an embodiment of the present invention. Referring to Fig. 5, a prefetch latch 510 is coupled to the program memory 500, an instruction register 520 and a holding register 540. The prefetch latch 510 holds the instruction fetched from program memory 500 at the location specified by the program counter (PC) 570. The instruction register 520 receives the instruction from the prefetch latch 510 upon the beginning of Q1 of the next instruction cycle. The holding register 540 receives and stores the value in the prefetch latch 510 when an interrupt is taken. The PC shadow register 571 receives and stores the contents of the PC when an interrupt is taken. The SR shadow register receives and stores the contents of the status register.

The instruction stored in the instruction register 520 is decoded and provided to the execution units as indicated in Fig. 5. When an interrupt is taken and to speed processing of the interrupt service routine (ISR), the first interrupt instruction from the ISR may be transferred from the first interrupt instruction register 530 to the instruction register 520. In addition, the PC is transferred to the PC shadow register and the status register to the SR shadow register. This is called a fast interrupt , permits the ISR to begin execution immediately, without having to wait for the processor 100 to fetch the first ISR instruction by first changing the program counter 570. This functionality is referred to as a fast interrupt processing or low overhead interrupt processing.

The program counter 570 typically increments during sequential instruction flow. It may also include loop control functionality for implementing repeat instruction processing according

to embodiments of the present invention. It may also include logic for processing branch and other flow control instructions. The interrupt logic 580 reads various inputs to determine whether an exception to normal instruction processing has occurred and whether the exception should trigger an interrupt of the normal instruction flow. There are many conditions that may trigger an interrupt including, for example, a processor reset, an oscillator fail condition, a stack overflow condition, an address error condition, an illegal instruction condition, an arithmetic error condition and various priority interrupts for giving effect to various input/output, program or other processes. A processor reset, however, will not result in saving the status of a repeat instruction.

The interrupt logic 580 determines the presence of an interrupt and whether or not to process the interrupt based on the methods shown in Figs. 5, 6A and 6B. If the interrupt logic 580 determines that an interrupt is to be taken, then the interrupt logic causes: a) the value in the prefetch latch 510 to be stored into the holding register 540 and b) if the interrupt is designated to be a fast interrupt, the first interrupt instruction to be loaded from the first interrupt instruction register 530 to the instruction register 520. The program counter 570 also loads into the program counter 570 the address of the first or second instruction of the ISR from the ISR vector register 560 depending on whether the interrupt is a fast interrupt or a regular interrupt. If the interrupt is a fast interrupt, the PC is also transferred into the PC shadow register. If the interrupt is not a fast interrupt, the interrupt logic 580 also causes the program counter 570 and the status register to be stored on the stack 550 prior to changing the program counter.

In this manner, the ISR interrupts the normal instruction execution and takes over execution for the duration of the ISR. When the ISR is finished, the program counter and the status register are restored from the stack or the shadow register, depending on the interrupt type.

Program processing resumes where it left off. Moreover, in the case of a fast interrupt, to speed the process of resuming processing, the next instruction for execution is restored from the holding register 540 into the prefetch latch 510 without requiring the next instruction to be fetched first from program memory based on the program counter 570. This eliminates wasted processor cycles. While the instruction from the holding register 540 is executing, the next instruction is fetched from the program memory based on a program counter value restored from the shadow register.

Fig. 6A depicts a method of performing interrupt processes 600 in an efficient manner according to an embodiment of the present invention. Referring to Fig. 6A, in step 605, the processor checks an interrupt status register to determine if there has been an interrupt. If the status register indicates that there has been no interrupt in step 610, then step 605 begins again. If the status register indicates that there has been an interrupt, then step 620 begins. In step 620, the processor arbitrates priority to determine the interrupt with the highest priority. This step is provided because there may be more than one interrupt with different priority levels at any given time. Then in step 625, the interrupt logic determines whether the interrupt with the highest priority has a priority level which exceeds that of the CPU priority. This step in essence determines whether the interrupt pending is more important than the process that the processor is currently running. If not, then step 605 begins again and the interrupt is not serviced. If so, then step 630 begins.

In step 630 the processor determines whether the interrupt is a fast interrupt or a regular interrupt. This step may be performed by comparing an interrupt identifier with a list of those interrupts specified as fast interrupts. Fast interrupts take advantage of the registers 530-560 to speed launch into the ISR and return from the ISR. Regular interrupts do not take advantage of

these registers. All interrupts use the stack 550. The processor may select one interrupt as a fast interrupt. In addition, the processor should store the following information for the interrupt specified as a fast interrupt: a) the first instruction of the ISR into the first interrupt instruction register 530, b) the address of the second ISR instruction into the ISR vector register. The values represented in a) and b) may be retrieved from memory based on the type of fast interrupt being serviced.

If a fast interrupt occurs, then step 635 begins which is illustrated in Fig. 6B. If a fast interrupt does not occur, then a regular interrupt is processed according to steps 640 and 645 shown in Fig. 6A. In step 640 the processor pushes the program counter and status register onto the stack. Then in step 645, the processor loads the address of the first instruction in the ISR into the program counter 570. Then in step 650, prefetch latch fetches the first ISR instruction with attendant delay and executes the ISR. At the end of the ISR, the program counter and status register are restored from the stack and regular instruction processing resumes.

When a fast interrupt is determined to have caused an interrupt, then the fast interrupt processes 635 begin as depicted in Fig. 6B. Referring to Fig. 6B, in step 700 the processor interrupt logic 580 loads the first interrupt instruction from the first interrupt instruction register 530 into the instruction register 520. Then in step 705, the processor causes the prefetch latch 510 to store the next instruction for regular execution into the holding register 540. The processor also causes the program counter 570 to be stored in the program counter shadow register 571 and the status register to be stored in the shadow register for the status register. In step 710, the address of the second ISR instruction is loaded from the ISR vector register into the program counter.

In step 715, the processor executes the instruction from the instruction register which is the first instruction of the ISR. This process occurs without loss of processing cycles associated with fetching the instruction based on the program counter. At the same time, in step 720, the processor fetches second ISR instruction into the prefetch latch 510 from the program memory based on the program counter 570. In step 725, the program counter is incremented as the instructions within the ISR are fetched and executed. At the end of the ISR, the interrupt logic causes a return from servicing the ISR.

In step 730, upon return from the ISR, the processor causes the instruction stored in the holding register 540 to be restored to the prefetch latch 510. This is the instruction that was about to be executed prior to the interrupt. Then in step 735, the value of the program counter prior to the interrupt is restored from the stack 550 or the shadow register into the program counter 570. The value of the status register prior to the interrupt is also restored from the stack into the status register.

In step 740, the processor executes the next instruction of the program flow prior to the ISR. The next instruction may be a repeat instruction or other instruction stored within prefetch latch 510 upon resuming normal program execution. The execution resumes without wasted processor cycles because the next instruction for execution is placed directly into the prefetch latch from the holding register for execution in the next processor cycle. If this were not the case execution would be delayed for at least one processor cycle while the processor fetched the next instruction from memory based on the restored program counter.

While specific embodiments of the present invention have been illustrated and described, it will be understood by those having ordinary skill in the art that changes may be made to those embodiments without departing from the spirit and scope of the invention.